Třídění II.

Základy algoritmizace – 10. cvičení

Zuzana Petříčková

26. dubna 2020

Třídění - pokračování

Úloha

- chceme setřídit (tj. uspořádat podle nějakého klíče) data uložená v poli / ve spojovém seznamu / v souboru
  - data chceme podle klíče uspořádat vzestupně (nebo sestupně)

Typy třídění

- **vnitřní** - všechna data jsou uložena ve vnitřní paměti počítače
- **vnější** - data jsou uložena na disku (např. se celá do paměti nevejdou)

My se na tomto cvičení budeme zabývat pouze vnitřním tříděním

- budeme třídit čísla uložená v poli či ve spojovém seznamu
- cílem bude uspořádat čísla vzestupně

Třídění

Třídící algoritmy můžeme rozdělit do tří kategorií:

- přímé metody (probírali jsme minule)
  - krátké a jednoduché algoritmy (obvykle založené na porovnávání)
  - třídí tzv. ,,na místě" (přímo v poli, používají jen konstantně velkou pomocnou paměť)
  - typicky časová složitost $O(n^2)$
  - typičtí představitelé:
    - **selection sort** = třídění přímým výběrem minima
    - **insertion sort** = třídění vkládáním
    - **bubble sort** = bublinkové třídění
    - a další (např. shell sort, třídění binárním vkládáním,...)
- sofistikovanější třídící algoritmy
- třídící algoritmy speciální, ,,ušité na míru" konkrétním datům

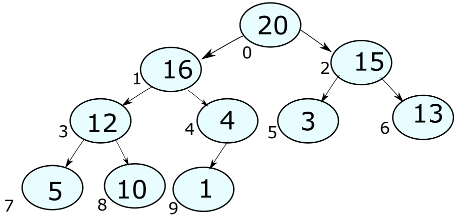
Sofistikovanější třídící algoritmy

- algoritmy náročnější na implementaci
- časová složitost $O(nlog_2 n)$ v průměru nebo vždy
- typickou cenou za nižší časovou složitost je vyšší prostorová složitost, obvykle využívají pomocnou paměť velikosti $O(n)$
- kdy se hodí? ... když chceme setřídit ,,hodně" dat
- typičtí představitelé:
    - **třídění binárním vyhledávacím stromem** (bylo minule)
    - **merge sort** = třídění přímým slučováním (bylo minule)
    - **heap sort** = třídění haldou
    - **quick sort** = rychlé třídění

Heap sort (třídění haldou)

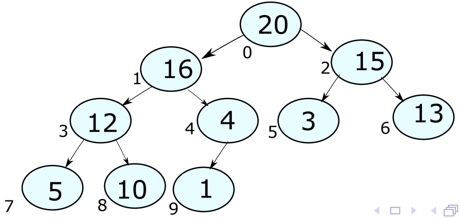
Základní myšlenka

- využíva při třídění pomocnou datovou strukturu (halda)
- pokud třídíme pole, může být halda uložená přímo v tříděném poli



Halda (heap)

- binární strom s následujícími vlastnostmi:
  - všechny úrovně jsou zcela zaplněny (až na poslední úroveň, ta je zaplněná ,,zleva")
  - hodnota každého vrcholu je větší nebo rovna hodnotám jeho synů (tzv. **MAX-halda**, existuje i MIN-halda s opačnou podmínkou)
  - více viz skripta, kapitola 2.2.8 (str. 58)
- haldu můžeme snadno implementovat v poli
  - synové vrcholu uloženého na indexu i budou na indexech $(2i + 1)$ a $(2i + 2)$
- hloubka haldy o n prvcích $\sim log_2 n$



Heap sort (třídění haldou)

Základní myšlenka algoritmu

1. z pole / seznamu postavíme MAX-haldu
2. z haldy postupně odebíráme kořen (=maximum) a vkládáme ho zpět do pole/seznamu

I. Heap sort pro spojový seznam:

```
void Seznam::heapSort()
{
    MAXHalda h;
    while (neprazny())
        h.vloz(vyjmiPrvni());
    while (h.neprazdna())
        vlozNaZacatek(h.vyjmiMax());
}
```

Heap sort (třídění haldou)

Pro haldu si musíme rozmyslet, jak implementovat dvě metody:

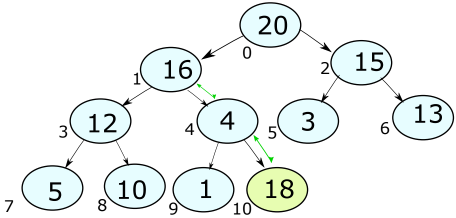
1. vlož (INSERT)
2. vyjmiMax (EXTRACTMAX)

Ukážeme si teď trochu jinou variantu, než je ve skriptech.

Heap sort (třídění haldou)

vlož (INSERT):

1. vlož nový prvek ,,na konec" haldy (na nejnižší úroveň na volné místo nejvíce vlevo)
2. probublávej nový prvek nahoru (dokud je otec menší, prohoď nový prvek s jeho otcem)

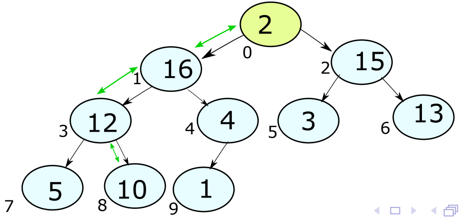


Heap sort (třídění haldou)

vyjmiMax (EXTRACTMAX):

1. maximum je v kořeni, po jeho vyjmutí by zůstal strom bez kořene
2. do kořene přesuneme prvek ,,z konce'' haldy (nejpravější prvek na nejnižší úrovni)
3. probublávej tento prvek dolů (dokud je alespoň jeden ze synů větší, prohoď prvek s jeho největším synem)

Složitost obou operací je $O(log_2 n)$ (procházíme vždy jen jednu větev stromu, a to nahoru / dolů)



Heap sort (třídění haldou)

Třídění pole čísel

- halda bude uložená přímo v tříděném poli
  - kořen haldy bude na indexu 0,
  - synové vrcholu na indexu i budou na indexech $2i + 1$ a $2i + 2$, jeho otec bude na indexu $\lfloor (i - 1)/2 \rfloor$

I. Postavíme haldu:

- haldu budeme postupně stavět zleva doprava
  - postupně bereme jeden prvek pole za druhým (pro $i = 1, ..., n - 1$) a zabubláme ho „zdola" do haldy (tvořené nyní prvky pole na indexech $0, .., i$)
- nakonec tvoří celé pole haldu

Heap sort (třídění haldou)

II. Z haldy sestrojíme setříděné pole:

- pak budeme postupně vyjímat z haldy maximum a halda se bude postupně zmenšovat (zprava bude postupně vznikat setříděné pole)
  - vždy vezmeme prvek na indexu 0 (maximum) a prohodíme ho s prvkem na indexu i, $i = n - 1, n - 2, ..., 1$). Prvek, který byl přesunut na index 0, zabubláme ,,shora" do haldy (tvořené prvky pole na indexech $0, ..., i - 1$)

Heap sort (třídění haldou)

Časová složitost

- složitost INSERT i EXTRACTMAX je $T(n) = O(log_2 n)$
- postavení haldy: $T(n) = nlog_2 n$
- vytvoření setříděného pole/seznamu : $T(n) = O(nlog_2 n)$
- celkem $T(n) = O(nlog_2 n)$

Prostorová složitost

- pro spojový seznam: $S(n) = O(n)$ (halda)
- pro pole: $S(n) = O(1)$ (halda je přímo v poli, třídíme tzv. ,,na místě")

Quick sort (rychlé třídění)

Základní myšlenka (pro pole čísel): metoda rozděl a panuj (ale jde na to trochu jinak než merge sort)

1. vezmeme libovolný prvek pole, nazveme ho **pivot**
2. **QSPLIT:** rozdělíme pole na tři části:
   - prvky menší nebo rovné pivotu (budou v poli vlevo od pivotu)
   - pivot
   - prvky větší nebo rovné pivotu (budou v poli vpravo od pivotu)
3. obě části pole setřídíme (např. rekurzivně nebo s použitím zásobníku)

QMERGE (slití setříděných polí) při implementaci ,,na místě'' není potřeba

Quick sort (rychlé třídění)

Jak vybrat pivot?

- např. jako prostřední prvek tříděného úseku pole (optimální varianta, pokud je pole již setříděné)
- nebo jako první prvek tříděného úseku pole

Jak rozdělit úsek pole x $[l, .., r]$ **podle pivotu?**

- např. zavedeme dva indexy $i = l$ a $j = r$ a budeme prohazovat moc velké prvky před pivotem s moc malými prvky za pivotem:
  1. dokud je $i \leq j$ a $x[i] \leq pivot$, zvyšujeme i o jedna
  2. dokud je $i \leq j$ a $x[j] \geq pivot$, snižujeme j o jedna
  3. pokud je $i < j$, prohodíme $x[i]$ a $x[j]$
  4. pokud je $i \leq j$, zvýšíme i o jedna a snížíme j o jedna
  5. celý proces opakujeme, dokud je $i \leq j$
- nakonec máme dva úseky pole $[l, ..., j]$ a $[i, ..., r]$ pro další třídění

Quick sort (rychlé třídění)

Časová složitost

- záleží na tom, jak dobře jsme zvolili pivot
- **nejlepší případ**: pivot = medián prvků
  $\rightarrow T(n) = 2T(n/2) + O(n) = O(nlog_2 n)$
  (na rozmyšlenou, použijte master theorem = kuchařku)
- **nejhorší případ**: pivot = nejmenší nebo největší prvek úseku
  $\rightarrow T(n) = 2T(n-1) + (n-1) = O(n^2)$
  (například: již setříděné pole a jako pivot bereme první prvek úseku)
- **průměrný případ**: $T(n) = O(nlog_2 n)$ (viz skripta)

Prostorová složitost

- **nejhorší případ**: $S(n) = O(n)$ (maximální hloubka rekurze nebo velikost zásobníku)
- **průměrný případ**: $S(n) = O(log_2 n)$

Quick sort (rychlé třídění)

Implementace pomocí zásobníku

- na zásobník budeme vkládat dvojice indexů (l, r) (indexy začátku a konce tříděného úseku pole)

```
if (n>0)
   vloz na zasobnik prvek (0,n-1)
while(zasobnik neni prazdny) {
        vyjmi vrchol zasobniku (l,r)
        i =l, j=r
        zvol pivot
        proved qsplit podle pivotu
        if (l < j)
             vloz na zasobnik prvek (l,j)
        if (i < r)
             vloz na zasobnik prvek (i,r)

    }
}
```

Spojový seznam ... quickSort (rekurzivně)

```
void Seznam::quickSort()
{
   if (prvni == zarazka || prvni->dalsi == zarazka)
      return;
   Seznam t, u;
   qsplit(t, u);
   t.quickSort();
   u.quickSort();
   qmerge(t, u);
}
```

Spojový seznam ... quickSort (zásobník) ... pseudokód

```
void Seznam::quickSort() {
   Seznam *s, *t, *u;
   Zasobnik z; //prvek = ukazatel na seznam nebo trojice ukazatelu
   if (asponDvouprvkovy())
      z.vloz(this);
   while(z.neprazdny()) {
      if (z.naVrchuJeden()){
           s = z.vyjmi();
           t = new Seznam(); u = new Seznam();
           s->qsplit(t, u);
           z.vloz(s,t,u);
           if (t.asponDvouprvkovy())
               z.vloz(t);
           if (u.asponDvouprvkovy())
               z.vloz(u);
      }
      else {
           (s,t,u) = z.vyjmi(); // zde pseudokod
           s->qmerge(t, u);
           delete t; delete u;
      }
}
```

Quick sort (rychlé třídění)

Hoarův algoritmus (metoda vyhledání k-tého nejmenšího prvku pole)

- algoritmus založený na podobném principu jako rychlé třídění
- abychom rychleji našli k-tý nejmenší prvek pole, není třeba pole třídit
- více viz. skripta, str. 150-152

Hoarův algoritmus

Základní myšlenka (pro pole čísel):

1. vezmeme libovolný prvek pole, nazveme ho **pivot**
2. **QSPLIT:** rozdělíme pole na tři úseky:
   - prvky menší nebo rovné pivotu (budou v poli vlevo od pivotu)
   - pivot
   - prvky větší nebo rovné pivotu (budou v poli vpravo od pivotu)
3. v prohledávání pokračujeme jen v tom úseku pole, kde se nachází index k

Hoarův algoritmus

Časová složitost

- záleží na tom, jak dobře jsme zvolili pivot
- **nejlepší případ**: pokud je k-tý prvek na svém správném místě, pak stačí jeden průchod polem, $T(n) = O(n)$
- **nejhorší případ**: pivotem zvolíme (k-1)-krát maximum a (n-k)-krát minimum pole (na rozmyšlenou) $\rightarrow T(n) = O(n^2)$

Přihrádkové třídění (bucket sort)

Zadání:

- máme setřídit prvky v poli podle nějakého klíče
- klíč přitom nabývá jen malého počtu hodnot (např. k), hodnoty se mohou opakovat
- metoda je popsaná ve skriptech, kapitola 5.5.1 (str. 166-167)

Základní myšlenka (pro pole čísel)

1. připravíme si k přihrádek (každá funguje jako fronta a odpovídá jednomu klíči)
2. prvky pole nasypeme do přihrádek (podle hodnoty klíče)
3. překopírujeme přihrádky jednu po druhé zpět do pole

Přihrádkové třídění (bucket sort)

Časová složitost

1. přesypání prvků do přihrádek: $T(n) = O(n + k)$
2. vysypání prvků z přihrádek: $T(n) = O(n)$
3. celkem: $T(n) = O(n + k)$

Prostorová složitost $S(n) = O(n + k)$ (na přihrádky)

Spojový seznam ... bucketSort

```
void Seznam::bucketSort(int k)
{
    Fronta f[k];
    while (neprazny())      // nasypej prvky seznamu do prihradek
    {
        Prvek *p = vyjmiZacatek();
        f[p.klic()].vloz(p);
    }
    for(int i = 1; i < k; i++) // vysypej prihradky zpet
        while (f[i].neprazdna())
            vlozNaKonec(f[i].vyjmi());
}
```

- implementace přihrádek pomocí fronty (ne např. zásobníku) má zásadní význam u lexikografického třídění a radixSortu → zajišťuje stabilitu

Přihrádkové třídění (bucket sort)

Speciální případy

- třídění čísel podle základu (radix sort)
  (skripta, kapitola 5.5.3 (str. 169-170))
- lexikografické třídění
  - přihrádkové třídění od posledního písmene po první
  - více viz skripta, kapitola 5.5.2 (str. 167-169)

Třídění čísel podle základu (radix sort)

- máme setřídit čísla o nichž víme, že mají omezený počet cifer (například jsou maximálně trojciferná)
- použijeme přihrádkové třídění pro jednotlivé cifry:
  1. nejprve pole setřídíme podle podle nejnižší cifry
  2. pak podle druhé nejnižší cifry
     ... (atd.)
  3. nakonec podle nejvyšší cifry
- **POZOR:** abychom si nepokazili to, co již máme setříděné, je důležité přihrádky implementovat jako fronty (ne třeba zásobníky)

Časová složitost

1. přihrádkové třídění podle jedné cifry (přihrádky 0-9): $T(n) = O(n + 10)$
2. Celkem: $T(n) = O(l(n + 10))$ (l je počet cifer)

Prostorová složitost $S(n) = O(n + 10)$ (na přihrádky)

Lexikografické třídění

- máme setřídit slova, věty ap. podle abecedy
- použijeme přihrádkové třídění pro jednotlivá písmena:
    1. nejprve slova setřídíme podle podle posledního písmene
    2. pak podle druhého písmene od konce

       ... (atd.)
    3. nakonec podle prvního písmene
- **POZOR:** abychom si nepokazili to, co již máme setříděné, je důležité přihrádky implementovat jako fronty (ne třeba zásobníky)
- **POZOR:** je třeba ošetřit situaci, kdy slova mají různou délku

Shrnutí: složitost různých třídících algoritmů

	MIN	AVERAGE	MAX	paměť (pro pole)
selectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
bubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
shellSort		$O(n^\alpha)$		$O(1)$
treeSort		$O(n log_2 n)$	$O(n^2)$	$O(n)$
mergeSort		$O(n log_2 n)$		$O(n)$
quickSort		$O(n log_2 n)$	$O(n^2)$	$\Omega(log_2 n) ... O(n)$
heapSort		$O(n log_2 n)$		$O(1)$
bucketSort		$O(n+k)$		$O(n+k)$

- n je délka pole/seznamu
- k je počet přihrádek
- $1 < \alpha < 2$